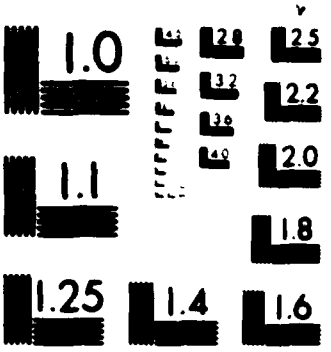
AD-A182 633 SYSTOLIC ALGORITHMS FOR THE PARALLEL SOLUTION OF DENSE SYMMETRIC POSITIVE.. (U) YALE UNIV NEW HAVEN CT DEPT OF COMPUTER SCIENCE I C IPSEN MAY 87 YALEU/DCS/RR-539 1/1

UNCLASSIFIED N00014-86-K-0310 F/G 12/6 NL

FN1
9-87
DTIC



Systolic Algorithms for the Parallel Solution of Dense Symmetric Positive-Definite Toeplitz Systems

Ilse C.F. Ipsen

Research Report YALEU/DCS/RR-539
May 1987

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

5

Abstract. The most popular method for the solution of linear systems of equations with Toeplitz coefficient matrix on a single processor is Levinson's algorithm, whose intermediate vectors form the Cholesky factor of the inverse of the Toeplitz matrix. However, Levinson's method is not amenable to efficient parallel implementation. In contrast, use of the Schur algorithm, whose intermediate vectors form the Cholesky factor of the Toeplitz matrix proper, makes it possible to perform the entire solution procedure on one processor array in time linear in the order of the matrix.

By means of the Levinson recursions we will show that all three phases of the Toeplitz system solution process: factorisation, forward elimination and backsubstitution, can be based on Schur recursions. This increased exploitation of the Toeplitz structure then leads to more efficient parallel implementations on systolic arrays.

To appear in: *Proceedings of the IMA Workshop on Numerical Algorithms for Modern Parallel Computer Architectures*, M.H. Schultz, ed., Springer Verlag, 1987.

Systolic Algorithms for the Parallel Solution of Dense Symmetric Positive-Definite Toeplitz Systems

Ilse C.F. Ipsen

Research Report YALEU/DCS/RR-539
May 1987

The work presented in this paper was supported by the Office of Naval Research under contracts N000014-86-K-0310 and N00014-85-K-0461, and by the Army Research Office under contract DAAL03-86-K-0158.

Introduction

The aim of this paper is to discuss parallel methods for the solution of linear systems of equations

$$T_n x = f$$

whose coefficient matrices T_n are dense symmetric positive-definite Toeplitz matrices.

A symmetric Toeplitz matrix $T_n = (t_{kl})_{0 \le k,l \le n}$ of order $n+1$ is a matrix whose elements are constant along each diagonal, $t_{kl} = t_{|k-l|}$. The solution of a general, $n \times n$ system of equations by a direct method requires $O(n^3)$ operations. Since a $n \times n$ Toeplitz matrix is characterised by $O(n)$ rather than $O(n^2)$ parameters efficient algorithms for the solution of Toeplitz systems exhibit an operation count that is considerably smaller: the classical Levinson and Schur algorithms require $O(n^2)$ operations [1, 14, 15, 18] while the doubling algorithms require $O(n \log^2 n)$, cf. the early references [2, 9]. A thorough treatment of Toeplitz matrices is given in [10, 11], and a brief summary can be found in [17]. Numerical aspects of algorithms for Toeplitz matrices are reviewed in [5].

Development of parallel implementations for the solution of dense Toeplitz systems was motivated by the need to execute certain signal processing tasks in real-time. The preferred architectures are *systolic arrays*, special-purpose devices built with Very Large Scale Integrated (VLSI) circuit technology [13]. Systolic arrays are homogeneous networks of tightly coupled, highly synchronised, simple processors that essentially operate in SIMD (Single Instruction Multiple Data stream) mode. Due to the repetitiveness of the computations and the regularity of the data dependencies systolic implementations can be described by means of linear transformations: the processor in which a quantity $r_{i,j}$ is computed as well as the time of its computation is expressed as a linear function in the indices i and j [7, 8]. To keep the approach simple and intuitive, implementation details will be omitted in this paper, they can be found in [7, 8].

Three classes of systolic arrays will be presented whose efficiency improves with increased exploitation of the Toeplitz structure in various phases of the solution process.

The classical method of choice for solving a $n \times n$ symmetric positive-definite Toeplitz system on a single processor is the Levinson algorithm [14]. The intermediate vectors generated by the Levinson algorithm form the Cholesky factor of the inverse of the Toeplitz matrix. Due to a sequence of n inner products, however, the lower bound of the parallel solution time on n processors is $O(n \log n)$.

The second classical method is the Schur algorithm [1, 15, 18]; its intermediate vectors form the Cholesky factor of the Toeplitz matrix. Although its operation count is fifty percent higher

than that of Levinson's method, it is more amenable to parallel implementation: an array of $O(n)$ processors can determine the Cholesky factor of an order-n Toeplitz matrix in time $O(n)$ [3, 4, 6, 7, 8, 12, 16].

The solution to the Toeplitz system can be found by performing forward elimination with the transpose of the Cholesky factor and subsequent backsubstitution involving the Cholesky factor. This necessitates the additional use of arrays for triangular system solution and intermediate storage of order $O(n^2)$ for the Cholesky factor during forward elimination [12, 16].

Instead of performing the usual forward elimination, recursions similar to the 'Schur recursions' in the factorisation may can be employed to modify the right-hand side vector, thus making it possible to employ the same type of array for factorisation and forward elimination. Intermediate storage of the Cholesky factor till the start of backsubstitution may be avoided by re-generating it on the fly [3, 4].

A final improvement in efficiency is achieved by also performing backsubstitution by Schur recursions. In this case it is possible to perform the whole solution process on one n-processor array in time $O(n)$ [7, 8].

Since it appears impossible to conceive systolic implementations of doubling algorihms, it can be concluded that the most efficient method hitherto to solve Toeplitz systems on systolic arrays is one that makes maximum use of Schur recursions.

Notation

A symmetric Toeplitz matrix of order $n+1$ will be denoted by T_n, where

$$T_n = \begin{pmatrix} t_0 & t_1 & \dots & \dots & t_n \\ t_1 & t_0 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & t_0 & t_1 \\ t_n & \dots & \dots & t_1 & t_0 \end{pmatrix},$$

and the sequence $t_i \dots t_k$ of Toeplitz matrix elements for $i \leq k$ will be denoted by $t_{i:k}$. Frequent use will be made of the fact that the first and last columns of T_n have the respective representations $t_{0:n}$ and $Jt_{0:n}$ where $J = (e_n \ \dots \ e_1 \ e_0)$ represents the 'exchange' matrix with ones on the antidiagonal, and e_i is a $(n+1) \times 1$ vector with a one in position i and zeros everywhere else, $0 \leq i \leq n$. At last, 0_k stands for the $k \times 1$ vector consisting of k zero elements; when $k = 0$ it is the empty vector.

A symmetric Toeplitz matrix T_n is centro-symmetric, that is

$$T_n = JT_nJ.$$

Direct methods with operation count $O(n^2)$ or less for the solution of dense Toeplitz systems of order n exploit this property and the resulting recursive nesting of Toeplitz matrices:

$$\left(\begin{array}{c|c} & t_n \\ T_{n-1} & \vdots \\ & t_1 \\ \hline t_n \quad \dots \quad t_1 & t_0 \end{array}\right) = T_n = JT_nJ = \left(\begin{array}{c|c} t_0 & t_1 \quad \dots \quad t_n \\ \hline t_1 & \\ \vdots & T_{n-1} \\ t_n & \end{array}\right).$$

The system $T_nx = f$ can be solved by recursively solving a system involving T_{n-1}.

The Levinson Algorithm

Levinson's algorithm computes the Cholesky factorisation of the inverse of a $n \times n$ Toeplitz matrices with $O(n^2)$ operations; the following derivation is partly based on the one given by Trench in [19]. Suppose the Cholesky factor L_n of $T_n^{-1} = L_n^T D_n^{-1} L_n$ is known, where L_n is unit lower triangular and D_n is diagonal (as T_n is symmetric positive-definite the diagonal elements of D_n are strictly positive). The Toeplitz matrix T_{n+1} of next higher order can be partitioned into a 2×2 block matrix as

$$T_{n+1} = \begin{pmatrix} T_n & t \\ t^T & t_0 \end{pmatrix}, \qquad t = Jt_{1:n+1} = \begin{pmatrix} t_{n+1} & \dots & t_1 \end{pmatrix}.$$

Solving

$$T_{n+1}^{-1} T_{n+1} = \begin{pmatrix} I_{n+1} & 0 \\ 0 & 1 \end{pmatrix},$$

where I_{n+1} is the identity matrix of order $n+1$, results in a 2×2 block partitioning for T_{n+1}^{-1}

$$T_{n+1}^{-1} = \begin{pmatrix} T_n^{-1} + T_n^{-1} t t^T T_n^{-1}/d & -T_n^{-1} t/d \\ -t^T T_n^{-1}/d & 1/d \end{pmatrix}.$$

Setting $\psi = -T_n^{-1} t$ yields

$$T_{n+1}^{-1} = \begin{pmatrix} T_n^{-1} + \psi\psi^T/d & \psi/d \\ \psi^T/d & 1/d \end{pmatrix} = \begin{pmatrix} I_n & \psi \\ 1 & \end{pmatrix} \begin{pmatrix} T_n^{-1} & \\ & d^{-1} \end{pmatrix} \begin{pmatrix} I_n & \\ \psi^T & 1 \end{pmatrix}$$

$$= \begin{pmatrix} L_n^T & \psi \\ 1 & \end{pmatrix} \begin{pmatrix} D_n^{-1} & \\ & d^{-1} \end{pmatrix} \begin{pmatrix} L_n & \\ \psi^T & 1 \end{pmatrix} = L_{n+1}^T D_{n+1}^{-1} L_{n+1}. \tag{1}$$

Thus, $(\psi^T \quad 1)$ is the trailing row of the Cholesky factor L_{n+1} of T_{n+1}^{-1}. It remains to show that ψ and d can be computed in $O(n)$ steps.

To this end, suppose that the Cholesky factorisation of T_n^{-1} is already known:

$$T_n^{-1} = L_n^T D_n^{-1} L_n = \begin{pmatrix} T_{n-1}^{-1} + \psi_n \psi_n^T / d_n & \psi_n / d_n \\ \psi_n^T / d_n & 1/d_n \end{pmatrix}, \qquad D_n = \begin{pmatrix} d_0 & & \\ & \ddots & \\ & & d_n \end{pmatrix},$$

where $d_0 = t_0$ and ψ_n is a $n \times 1$ vector. The symmetry-centro symmetry of Toeplitz matrices implies for the inverse of the next higher-order matrix T_{n+1} that $T_{n+1}^{-1} = J T_{n+1}^{-1} J$, or in block form

$$\begin{pmatrix} T_n^{-1} + \psi_{n+1}\psi_{n+1}^t / d_{n+1} & \psi_{n+1}/d_{n+1} \\ \psi_{n+1}^T / d_{n+1} & 1/d_{n+1} \end{pmatrix} = \begin{pmatrix} 1/d_{n+1} & \psi_{n+1}^T J / d_{n+1} \\ J\psi_{n+1}/d_{n+1} & T_n^{-1} + J\psi_{n+1}\psi_{n+1}^t J / d_{n+1} \end{pmatrix}. \tag{2}$$

This gives for the trailing row $\psi_{n+1} = -T_n^{-1} J t_{1:n+1}$ of L_{n+1} the block form

$$\begin{pmatrix} -1/d_n & -\psi_n^T J / d_n \\ -J\psi_n / d_n & -T_{n-1}^{-1} - J\psi_n \psi_n^T J / d_n \end{pmatrix} \begin{pmatrix} t_{n+1} \\ J t_{1:n} \end{pmatrix} = \begin{pmatrix} -(t_{n+1} + \psi_n^T t_{1:n})/d_n \\ -T_{n-1}^{-1} J t_{1:n} - J\psi_n (t_{n+1} + \psi_n^T t_{1:n})/d_n \end{pmatrix}.$$

Denoting the term in brackets by

$$\rho_{n+1} = -(t_{n+1} + \psi_n^T t_{1:n})/d_n = -(\psi_n^T \quad 1)\, t_{1:n+1}/d_n, \qquad \rho_1 = -t_1/t_0, \tag{3}$$

and observing that $\psi_n = -T_{n-1}^{-1} J t_{1:n}$ gives

$$\psi_{n+1} = \begin{pmatrix} \rho_{n+1} \\ \psi_n + \rho_{n+1} J \psi_n \end{pmatrix}.$$

Consequently, with ψ_0 the empty vector, the trailing row of L_{n+1} can be obtained from the trailing row of L_n via

$$\begin{pmatrix} \psi_{n+1} \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ \psi_n \\ 1 \end{pmatrix} + \rho_{n+1} \begin{pmatrix} 1 \\ J\psi_n \\ 0 \end{pmatrix}. \tag{4}$$

Remembering that d_n^{-1} is the bottom right element of T_n^{-1} and ρ_{n+1} is the leading element of ψ_{n+1} one gets with (2) for the bottom right element of T_{n+1}^{-1}

$$d_{n+1}^{-1} = d_n^{-1} + \rho_{n+1}^2 d_{n+1}^{-1} \quad \text{or} \quad d_{n+1} = d_n(1 - \rho_{n+1}^2), \qquad d_0 = t_0. \tag{5}$$

Note that the original paper by Levinson [14] does not contain the simple recursive computation of d_{n+1} from d_n and ρ_{n+1}.

The Levinson Algorithm

The Levinson algorithm computes the lower triangular Cholesky factor L_n of T_n^{-1} with kth row given by $(\psi_{k,0} \quad \cdots \quad \psi_{k,k-1} \quad 1 \quad 0_{n-k}^T)$.

$$d_0 = t_0$$

$$1 \le k \le n, \quad \rho_k = -(t_k + \sum_{j=1}^{k-1} \psi_{k-1,j-1} t_j)/d_{k-1}, \quad d_k = d_{k-1}(1 - \rho_k^2), \quad \psi_{k,0} = \rho_k$$

$$(\psi_{k,0} \quad \cdots \quad \psi_{k,k-1}) = (1 \quad \rho_k) \begin{pmatrix} \psi_{k-1,0} & \cdots & \psi_{k-1,k-2} \\ \psi_{k-1,k-2} & \cdots & \psi_{k-1,0} \end{pmatrix}$$

The reason why Levinson's algorithm has little potential for parallelisation is that the vector ψ_n enters into the computation of ψ_{n+1} from *both* ends: in a linear combination of ψ_n and $J\psi_n$. Even if one were to maintain two separate copies, ψ_n and $J\psi_n$, the weight ρ_{n+1} in this combination would still depend on the *entire* vector ψ_n. Thus, it is not possible to pipeline successive recursions, and the lower bound on the time of a $n \times n$ parallel Cholesky factorisation of is $O(n \log n)$.

The Schur Algorithm

Since the inner-product (3) in the formation of ρ_{n+1} is the culprit for the poor parallel performance of Levinson's algorithm one could try to reformulate the algorithm so as to obviate the need for an explicit inner-product computation. This is accomplished by observing that (1) implies $d_n = (\psi_n^T \quad 1)\, J t_{0:n}$, and substituting this into (3) leads to

$$\rho_{n+1} = -\frac{(\psi_n^T \quad 1)\, t_{1:n+1}}{(\psi_n^T \quad 1)\, J t_{0:n}}.$$

Thus, the coefficient ρ_{n+1} is the ratio of two quantities that are obtained by multiplying $(\psi_n^T \quad 1)$ and its reverse by a column of the Toeplitz matrix. This is the basis for the so-called Schur algorithm [1, 15, 18], it avoids the inner-product by recursively 'updating' matrix-vector products involving the Toeplitz matrix, so that ρ_{n+1} can be formed as the ratio of two vector elements.

Unlike the Levinson algorithm which determines the Cholesky factor of the inverse, the Schur algorithm determines the Cholesky decomposition of the matrix proper. If the result of Levinson's method is $T_n^{-1} = L_n^T D_n^{-1} L_n$, where L_n is lower triangular, then the uniqueness of the Cholesky

decomposition implies that $L_n T_n = D_n L_n^{-T}$ must be a scaled version of the upper triangular Cholesky factor U_n of T_n: $T_n = U_n^T D_n U_n$. Therefore it follows that $((\psi_k^T \quad 1) \quad 0_{n-k}^T)T_n$ is the kth row of the scaled Cholesky factor $D_n U_n$ of T_n. Let 0_k^T represent a row vector of k zeros, then the kth row of $D_n U_n$ has the form

$$((\psi_k^T \quad 1) \quad 0_{n-k}^T)T_n = ((\psi_k^T \quad 1) \quad 0_{n-k-1}^T \quad 0)\begin{pmatrix} T_k & A_k & * \\ A_k^T & T_{n-k-2} & * \\ * & * & t_0 \end{pmatrix}$$

$$= ((\psi_k^T \quad 1)T_k \quad (\psi_k^T \quad 1)A_k \quad *) = (0_k^T \quad d_k \quad (\psi_k^T \quad 1)A_k \quad *)$$
$$= (0_k^T \quad d_k \quad \nu_{k,0} \quad \dots \quad \nu_{k,n-k-1}), \tag{6}$$

where $*$ denotes unimportant terms and from (1)

$$(\psi_k^T \quad 1)T_k = (0_k^T \quad d_k). \tag{7}$$

If it is possible to compute the non-zero elements $(d_k \quad \nu_{k,0} \quad \dots \quad \nu_{k,n-k-1})$ of the kth row of $D_n U_n$ with $O(n-k)$ operations then the Cholesky factorisation $T_n = U_n^T D_n U_n$ can be computed with $O(n^2)$ operations. It is now shown how to compute the vector of $\nu_{k,j}$ as a linear combination of two vectors by making use of the Levinson recursion as follows:

$$((\psi_{k+1}^T \quad 1) \quad 0_{n-k-1}^T)T_n = (0 \quad (\psi_k^T \quad 1) \quad 0_{n-k-1}^T)T_n + \rho_{k+1}((\psi_k^T \quad 1)J \quad 0 \quad 0_{n-k-1}^T)T_n.$$

With (3), (7) and (6) the first summand evaluates to

$$(0 \quad (\psi_k^T \quad 1) \quad 0_{n-k-1}^T)\begin{pmatrix} t_0 & t_{1:k+1}^T & * \\ t_{1:k+1} & T_k & A_k \\ * & A_k^T & T_{n-k-2} \end{pmatrix}$$

$$= ((\psi_k^T \quad 1)t_{1:k+1} \quad (\psi_k^T \quad 1)T_k \quad (\psi_k^T \quad 1)A_k) = (-\rho_{k+1}d_k \quad 0_k^T \quad d_k \quad (\psi_k^T \quad 1)A_k)$$
$$= (-\rho_{k+1}d_k \quad 0_k^T \quad d_k \quad \nu_{k,0} \quad \dots \quad \nu_{k,n-k-2}).$$

The second summand amounts to

$$((\psi_k^T \quad 1)J \quad 0 \quad 0_{n-k-1}^T)\begin{pmatrix} T_k & Jt_{1:k+1} & B_k \\ t_{1:k+1}^T J & t_0 & * \\ B_k^T & * & T_{n-k-2} \end{pmatrix}$$

$$= ((\psi_k^T \quad 1)JT_k \quad (\psi_k^T \quad 1)t_{1:k+1} \quad (\psi_k^T \quad 1)JB_k) = (d_k \quad 0_k^T \quad -\rho_{k+1}d_k \quad (\psi_k^T \quad 1)JB_k)$$
$$= (d_k \quad 0_k^T \quad \mu_{k,0} \quad \dots \quad \mu_{k,n-k-1}).$$

where the leading element of the non-zero part is $\mu_{k,0} = -\rho_{k+1}d_k$.

Forming the linear combination of the two summands yields

$$\begin{aligned}
&((\psi_{k+1}^T \quad 1) \quad 0_{n-k-1}^T)T_n \\
&\quad = (-\rho_{k+1}d_k \quad 0_{n-k}^T \quad d_k \quad \nu_{k,0} \quad \cdots \quad \nu_{k,n-k-2}) + \rho_{k+1}(d_k \quad 0_{n-k}^T \quad \mu_{k,0} \quad \cdots \quad \mu_{k,n-k-1}) \\
&\quad = (0_{k+1}^T \quad d_{k+1} \quad \nu_{k+1,0} \quad \cdots \quad \nu_{k+1,n-(k+1)-1})
\end{aligned}$$

where due to (5) $d_{k+1} = d_k(1-\rho_{k+1}^2)$. Similarly, determination of the new vector elements $\mu_{k+1,i}$ is accomplished using the row-reversed version of (4)

$$\begin{aligned}
&((\psi_{k+1}^T \quad 1)J \quad 0_{n-k-1}^T)T_n \\
&\quad = ((\psi_k^T \quad 1)J \quad 0 \quad 0_{n-k-1}^T)T_n + \rho_{k+1}(0 \quad (\psi_k^T \quad 1) \quad 0_{n-k-1}^T)T_n \\
&\quad = (d_k \quad 0_k^T \quad \mu_{k,0} \quad \cdots \quad \mu_{k,n-k-1}) + \rho_{k+1}(-\rho_{k+1}d_k \quad 0_k^T \quad d_k \quad \nu_{k,0} \quad \cdots \quad \nu_{k,n-k-2}) \\
&\quad = (d_{k+1} \quad 0_{k+1}^T \quad \mu_{k+1,0} \quad \cdots \quad \mu_{k,n-(k+1)-1}),
\end{aligned}$$

where $\mu_{k+1,0} = -\rho_{k+2}d_{k+1}$.

The Schur Algorithm

The Schur algorithm computes the scaled Cholesky factor D_nU_n of T_n with kth row given by $(0_k^T \quad d_k \quad \nu_{k,0} \quad \cdots \quad \nu_{k,n-k-1})$.

$$d_0 = t_0$$

$$\begin{pmatrix} \nu_{0,0} & \cdots & \nu_{0,n-1} \\ \mu_{0,0} & \cdots & \mu_{0,n-1} \end{pmatrix} = \begin{pmatrix} t_1 & \cdots & t_n \\ t_1 & \cdots & t_n \end{pmatrix}$$

$$1 \le k \le n, \quad \rho_k = -\mu_{k-1,0}/d_{k-1}, \quad d_k = d_{k-1}(1-\rho_k^2)$$

$$\begin{pmatrix} \nu_{k,0} & \cdots & \nu_{k,n-k-1} \\ \mu_{k,0} & \cdots & \mu_{k,n-k-1} \end{pmatrix} = \begin{pmatrix} 1 & \rho_k \\ \rho_k & 1 \end{pmatrix} \begin{pmatrix} \nu_{k-1,0} & \cdots & \nu_{k-1,n-(k-1)-2} \\ \mu_{k-1,1} & \cdots & \mu_{k-1,n-(k-1)-1} \end{pmatrix}.$$

First Class of Systolic Implementations

Assume that a linear array of $n+1$ processors, numbered 0 to n, is available for the execution of the Schur algorithm on matrix T_n of order $n+1$. A time step for the array is defined as a time interval long enough to accommodate the operations

$$\rho_k = -\mu_{k-1,0}/d_{k-1}, \qquad d_k = d_{k-1}(1-\rho_k^2)$$

$$\begin{pmatrix} \nu_{k,j} \\ \mu_{k,j} \end{pmatrix} = \begin{pmatrix} 1 & \rho_k \\ \rho_k & 1 \end{pmatrix} \begin{pmatrix} \nu_{k-1,j} \\ \mu_{k-1,j+1} \end{pmatrix}.$$

Different schedules and processor assignments for individual operations can be derived by applying appropriate linear transformations to the indices of the computed quantities: the pair $(\nu_{k,j} \quad \mu_{k,j})$ is computed in processor

$$\pi = (k \quad j)\begin{pmatrix}\pi_1\\ \pi_2\end{pmatrix} + \pi_3$$

at time

$$\tau = (k \quad j)\begin{pmatrix}\tau_1\\ \tau_2\end{pmatrix} + \tau_3.$$

The partial order of computations is preserved by observing that $\nu_{k-1,j}$ must be computed before $\nu_{k,j}$, and $\mu_{k-1,j+1}$ before $\mu_{k,j}$. That is, the time function must satisfy

$$-\tau_1 < 0, \qquad -\tau_1 + \tau_2 < 0.$$

In general, if a quantity with index (i,j) depends on a quantity with index (k,l) the latter must be available before the former can be determined, in other words [8]

$$(k-i \quad l-j)\begin{pmatrix}\tau_1\\ \tau_2\end{pmatrix} < 0.$$

To make sure that one processor does not have to perform two different operations at the same time the determinant of the matrix

$$\begin{pmatrix}\tau_1 & \pi_1\\ \tau_2 & \pi_2\end{pmatrix}$$

should be non-zero [8], $\tau_1\pi_2 - \tau_2\pi_1 \neq 0$.

The first systolic array presented in [12] is based on the following linear transformations. In step k of the Schur algorithm, $0 \leq k \leq n$, $(\nu_{k,j} \quad \mu_{k,j})$ is computed in processor π at time $\tau \geq 1$ where

$$\pi = (k \quad j)\begin{pmatrix}0\\ 1\end{pmatrix} + 0 = j, \quad \tau = (k \quad j)\begin{pmatrix}1\\ 0\end{pmatrix} + 1 = k+1, \qquad 0 \leq j \leq n-k-1.$$

The parameters ρ_k and d_k are assumed to be associated with index $(k,0)$ and thus determined in processor $\pi = 0$ at time $\tau = k+1$. Initially processor j is loaded with matrix element t_j.

The execution of the Schur algorithm requires $n+1$ steps. In each step processor 0 computes a new ρ and broadcasts it to all other processors, so that each step produces a new row of the Cholesky factor. The components of the ν-vectors remain in their respective processors ($\nu_{k,j}$ resides in processor j for all k) while the μ-vector is shifted left by one in each step ($\mu_{k-1,j+1}$ is computed

in processor $j+1$ before being sent to processor j for the computation of $\mu_{k,j}$). Note that in step k there are k idle processors, and in order to offload a row of the Cholesky factor each processor must be able to perform external I/O.

To avoid difficulties, such as synchronisation delays and long wires, associated with a global communication scheme like broadcasting a 'pipelined' array is proposed in [12, 16]. The processor function $(\pi_1 \quad \pi_2 \quad \pi_3)$ is the same as before but the time function has changed to

$$(\tau_1 \quad \tau_2 \quad \tau_3) = (2 \quad 1 \quad 1).$$

Thus, ρ_k and d_k are computed at time $\tau = 2k+1$ in processor 0 and ρ_k is then sent to processor 1. In the next step, at $\tau = 2k+2$, $(\mu_{k,1} \quad \nu_{k,1})$ can be computed in processor 1, ρ_k can be transmitted to processor 2 and $\mu_{k,1}$ left to processor 1 so that at $\tau = 2k+3$ the computation of ρ_{k+1} can start. Thus, successive iterations are two time steps apart. Because ρ_n and d_n are computed at $\tau = 2n+1$ the computation time for the Schur algorithm comes to $2(n+1)$. The replacement of broadcasting by forwarding (or pipelining) of ρ from processor to processor results in communication that takes place exclusively on a nearest neighbour basis. All other features are the same as in the first array.

In order to *solve* the system $T_n x = f$ three possibilities are discussed in [12]:

1. forward elimination and backsubstitution involving the Cholesky factors U_n, D_n and U_n^T of T_n
2. computation of the Levinson vectors ψ_k using the ρ_k from the Schur algorithm, and subsequent matrix vector multiplications involving L_n, D_n, and L_n^T (the ψ_k constitute the rows of the Cholesky factor L_n of T_n^{-1})
3. explicit computation of T_n^{-1} in form of the Gohberg-Semencul formula and subsequent matrix-vector multiplications by means of FFTs (the Gohberg-Semencul formula represents the inverse of a Toeplitz matrix as a sum of products of triangular Toeplitz matrices that consist of the elements of ψ_n).

Since details for parallel implementations of the latter two methods are not given and their data and control flows are likely to be rather complex only the first method will be considered.

Forward Elimination with Cholesky Factor

Forward elimination solves the lower triangular system $(D_n U_n)^T h = f$, the elements of the solution vector h are given by $h_k = h_{k,k}$.

$$h_{0,0} = f_0/d_0$$

$$1 \le k \le n, \quad h_{k,-1} = 0$$

$$1 \le j \le k-1, \quad h_{k,j} = h_{k,j-1} + \nu_{j-1,k-j} h_{j,j}$$

$$h_{k,k} = (f_k - h_{k,k-1})/d_k.$$

In order to overlap forward elimination as much as possible with factorisation a second linear array with $n+1$ processors is employed, and it is assumed that each processor in the elimination array is physically connected to the corresponding processor in the factorisation array. Since a matrix element $\nu_{j-1,k-j}$ is computed in processor $k-j$ at time $\tau = k + j - 2$ in the factorisation array it may be used at the next time step in processor $k-j$ of the forward elimination array. Hence, the forward elimination array has the processor function

$$(\pi_1 \quad \pi_2 \quad \pi_3) = (1 \quad -1 \quad 0),$$

and essentially the same time function as the factorisation array (the time functions just differ by one in their displacement τ_3):

$$(\tau_1 \quad \tau_2 \quad \tau_3) = (2 \quad 1 \quad 2).$$

Note that the elements of the h-vector are shifted one processor to the right each step. At time $\tau = 3k + 2$, f_k and d_k have to be input to processor 0 of the elimination array so that $h_k = h_{k,k}$ can be computed there. Thus, forward elimination is completed after the computation of h_n at time $3n + 3$.

Backsubstitution with Cholesky Factor

Backsubstitution determines the solution x, with elements $x_k = x_{k,k}$, of the upper triangular system $D_n U_n x = D_n h$.

$$x_{n,n} = d_n h_n / d_n$$

$$n-1 \geq k \geq 0, \quad x_{k,n+1} = 0$$

$$n \geq j \geq k+1, \quad x_{k,j} = x_{k,j+1} + \nu_{k,j-k-1} x_{j,j}$$

$$x_{k,k} = (d_k h_k - x_{k,k+1})/d_k.$$

As backsubstitution can only start once forward elimination is completely finished, the forward elimination array may be re-used. Its time and processor functions are now

$$(\pi_1 \quad \pi_2 \quad \pi_3) = (-1 \quad 1 \quad 0), \quad (\tau_1 \quad \tau_2 \quad \tau_3) = (-1 \quad -1 \quad 5n+3).$$

If processor 0 has retained all h_k and d_k from the forward elimination phase then the solution element $x_k = x_{k,k}$ can be determined in processor 0 at time $\tau = 5n + 3 - 2k$. Note that solution of a Toeplitz system of order n in such a manner requires time $O(5n)$ on $2n$ processors plus $O(n^2)$ storage to store the matrix $D_n U_n$ during the forward elimination phase.

Forward Elimination by Schur Recursions

The second step, the modification of the right-hand side vector f in the system $T_n x = f$ can be improved for a systolic implementation by applying the same operations to f as were applied to T_n in the Schur algorithm: after having determined $L_n T_n = D_n U_n$ one now determines $g = L_n f$ so processors perform the same type of operations during facorisation and forward elimination, and only one type of array is needed for both phases.

To derive the computational steps for $g = L_n f$ we extend the vector f to a Toeplitz matrix F_n with f as its first column:

$$f = \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_n \end{pmatrix}, \qquad F_n = \begin{pmatrix} f_0 & f_1 & \cdots & \cdots & f_n \\ f_1 & f_0 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & f_0 & f_1 \\ f_n & \cdots & \cdots & f_1 & f_0 \end{pmatrix}.$$

From the computation of $L_n F_n$ one can derive recursions for $L_n f$ by means of the following obeservation. The kth element of $g = L_n f$ is

$$g_k = ((\psi_k^T \quad 1) \quad 0_{n-k}^T) f = (\psi_k^T \quad 1) f_{0,k}, \qquad 0 \le k \le n,$$

while the kth row of $L_n F_n$ is

$$((\psi_k^T \quad 1) \quad 0_{n-k}^T) F_n = ((\psi_k^T \quad 1) \quad 0_{n-k}^T) \begin{pmatrix} F_k & * \\ * & F_{n-k-1} \end{pmatrix}$$
$$= (* \quad (\psi_k^T \quad 1) F_k)$$

whose kth element is the trailing element of $(\psi_k^T \quad 1) F_k$ which is equal to $(\psi_k^T \quad 1) J f_{0,k}$. Hence, the trailing element of $(\psi_k^T \quad 1) J F_k$ in

$$((\psi_k^T \quad 1) J \quad 0_{n-k}^T) F_n = ((\psi_k^T \quad 1) J F_k \quad *)$$

is $g_k = (\psi_k^T \quad 1) f_{0,k}$.

Consequently, the sought vector g is a product of f with the matrix whose rows contain the reverse Levinson vectors, and g can be computed by the following Schur-like recursions involving the upper triangular part of this matrix.

Denote elements in position $k \le i \le n$ of

$$((\psi_k^T \quad 1) J \quad 0_{n-k}^T) F_n = ((\psi_k^T \quad 1) J \quad 0_{n-k}^T) \begin{pmatrix} F_k & C_k \\ C_k^T & F_k \end{pmatrix} = ((\psi_k^T \quad 1) J F_k \quad (\psi_k^T \quad 1) J C_k)$$

by

$$(\alpha_{k,k} \quad \cdots \quad \alpha_{k,n}) = ((\psi_k^T \quad 1) f_{0,k} \quad (\psi_k^T \quad 1) J C_k)$$

and elements in position $k \le i \le n$ of

$$((\psi_k^T \quad 1) \quad 0_{n-k}^T) F_n = ((\psi_k^T \quad 1) \quad 0_{n-k-1}^T \quad 0) \begin{pmatrix} F_k & D_k & J f_{n,n-k} \\ D_k^T & F_{n-k-2} & * \\ f_{n,n-k}^T J & * & f_0 \end{pmatrix}$$
$$= ((\psi_k^T \quad 1) F_k \quad (\psi_k^T \quad 1) D_k \quad (\psi_k^T \quad 1) J f_{n,n-k})$$

by

$$(\beta_{k,k} \quad \cdots \quad \beta_{k,n}) = ((\psi_k^T \quad 1) J f_{0,k} \quad (\psi_k^T \quad 1) D_k \quad (\psi_k^T \quad 1) J f_{n,n-k}) \tag{8}$$

Now $\alpha_{k,k} = g_k$ is the kth element of g and the recursive computation of $\alpha_{k+1,k+1} \quad g_{k+1}$ from $\alpha_{k,i}$ and $\beta_{k,i}$ can be derived by means of the Levinson recursions (4) as follows

$$((\psi_{k+1}^T \quad 1) \quad 0_{n-k-1}^T) F_n = ((\psi_k^T \quad 1) J \quad 0_{n-k}^T) + \rho_{k+1} (0 \quad (\psi_k^T \quad 1) \quad 0_{n-k-1}^T) F_n$$

Ignoring elements in positions 0 through k on both sides of the equation gives

$$(\alpha_{k+1,k+1} \quad \cdots \quad \alpha_{k+1,n}) = (\alpha_{k,k+1} \quad \cdots \quad \alpha_{k,n}) + \rho_{k+1}(\beta_{k,k} \quad \cdots \quad \beta_{k,n-1})$$

since the second summand is equal to

$$(0 \quad (\psi_k^T \quad 1) \quad 0_{n-k-1}^T) \begin{pmatrix} f_0 & f_{1:k}^T & * \\ f_{1:k} & F_k & D_k \\ * & D_k^T & F_{n-k-2} \end{pmatrix} = ((\psi_k^T \quad 1) f_{1:k} \quad (\psi_k^T \quad 1) F_k \quad (\psi_k^T \quad 1) D_k).$$

Comparing elements in positions $k+1$ through n with (8) one notes that

$$((\psi_k^T \quad 1) J f_{0:k} \quad (\psi_k^T \quad 1) D_k) = (\beta_{k,k} \quad \ldots \beta_{k,n-1}).$$

The second vector consisting of elements $\beta_{k+1,i}$, $k+1 \leq i \leq n$, can be updated similarly.

Forward Elimination by Schur Recursions

The Schur recursions determine $g = L_n f$ where $g_k = \alpha_{k,k}$.

$$\begin{pmatrix} \alpha_{0,0} & \cdots & \alpha_{0,n} \\ \beta_{0,0} & \cdots & \beta_{0,n} \end{pmatrix} = \begin{pmatrix} f_0 & \cdots & f_n \\ f_0 & \cdots & f_n \end{pmatrix}$$

$$1 \leq k \leq n, \qquad \begin{pmatrix} \alpha_{k,k} & \cdots & \alpha_{k,n} \\ \beta_{k,k} & \cdots & \beta_{k,n} \end{pmatrix} = \begin{pmatrix} 1 & \rho_k \\ \rho_k & 1 \end{pmatrix} \begin{pmatrix} \alpha_{k-1,k} & \cdots & \alpha_{k-1,n} \\ \beta_{k-1,k-1} & \cdots & \beta_{k-1,n-1} \end{pmatrix}.$$

Second Class of Systolic Implementations

Again, the same assumptions as before hold, and the array from [3, 4] is presented that performs both phases, factorisation and forward elimination, by Schur recursions.

The pipelined version of the factorisation phase is performed as before. As for forward elimination, the index structure of its equations is adapted to that of the factorisation by performing a linear transformation on each index $(k \quad j)$:

$$(k \quad j) \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} = (k \quad j-k),$$

resulting in

$$\begin{pmatrix} \alpha_{0,0} & \cdots & \alpha_{0,n} \\ \beta_{0,0} & \cdots & \beta_{0,n} \end{pmatrix} = \begin{pmatrix} f_0 & \cdots & f_n \\ f_0 & \cdots & f_n \end{pmatrix}$$

$$1 \le k \le n, \quad \begin{pmatrix} \alpha_{k,0} & \cdots & \alpha_{k,n-k} \\ \beta_{k,0} & \cdots & \beta_{k,n-k} \end{pmatrix} = \begin{pmatrix} 1 & \rho_k \\ \rho_k & 1 \end{pmatrix} \begin{pmatrix} \alpha_{k-1,1} & \cdots & \alpha_{k-1,n-(k-1)} \\ \beta_{k-1,0} & \cdots & \beta_{k-1,n-(k-1)+1} \end{pmatrix}.$$

If a second $(n+1)$-processor array is available for forward elimination with the same time and processor functions as those of the factorisation array, then factorisation and forward elimination can be performed simultaneously. Processor 0 of the forward elimination array is assumed to be connected to processor 0 of the factorisation array so the latter can forward the ρ_k to the former.

When pipelining is used, processor 0 of the elimination array receives ρ_k and forwards it directly to the other processors in the array so pairs $(\alpha_{k,j} \quad \beta_{k,j})$ are computed in processor j at time $2k+j+1$. Initially, processor j is loaded with the right-hand element f_j. Element $g_k = \alpha_{k,0}$ is computed in processor 0 at time $\tau = 2k+1$ and transmitted to processor 0 of the factorisation array where it is retained till the start of the backsubstitution phase. Thus, factorisation and forward elimination can be executed on $2(n+1)$ processors in $2(n+1)$ time steps if communication proceeds on a nearest neighbour basis.

In order to avoid the $O(n^2)$ storage needed to store D_nU_n till the onset of backsubstitution only its last column and the parameters ρ_k are retained from which D_nU_n can be re-generated by the Schur recursions.

The Reverse Version of the Schur Algorithm

The reverse version of Schur algorithm computes the scaled Cholesky factor D_nU_n of T_n with kth row given by $(0_k^T \quad d_k \quad \nu_{k,0} \quad \cdots \quad \nu_{k,n-k-1})$ from ρ_k, $1 \le k \le n$, and the last column

$$(\nu_{0,n-1} \quad \nu_{1,n-2} \quad \cdots \quad \nu_{n-1,0} \quad d_n)^T$$

of D_nU_n, whereby $\nu_{0,n-1} = t_n$.

$$n-1 \ge k \ge 0,$$

$$\begin{pmatrix} \nu_{k,0} & \cdots & \nu_{k,n-k-2} \\ \mu_{k,1} & \cdots & \mu_{k,n-k-1} \end{pmatrix} = \frac{1}{\rho_{k+1}^2 - 1} \begin{pmatrix} -1 & \rho_{k+1} \\ \rho_{k+1} & -1 \end{pmatrix} \begin{pmatrix} \nu_{k+1,0} & \cdots & \nu_{k+1,n-(k+1)-1} \\ \mu_{k+1,0} & \cdots & \mu_{k+1,n-(k+1)-1} \end{pmatrix}$$

$$d_k = d_{k+1}/(1-\rho_{k+1}^2), \quad \mu_{k,0} = -d_k\rho_{k+1}.$$

If processor 0 in the factorisation array has retained all ρ_k and d_n, and processor $n-k$ has received component $\nu_{k,n-k-1}$ of the last column from its left neighbour then the re-generation of D_nU_n in the factorisation array can start at time $2(n+1)$. The processor and time functions are

$$(\pi_1 \quad \pi_2 \quad \pi_3) = (0 \quad 1 \quad 0), \quad (\tau_1 \quad \tau_2 \quad \tau_3) = (-2 \quad -1 \quad 4n+2),$$

so that $(\nu_{k,j} \quad \mu_{k,j+1})$ is computed in processor j at time $4n+2-2k-j$, and d_k in processor 0 at time $\tau = 4n+2-2k$. Processor 0 stores the d_k for the backsubstitution phase. Note that the components of the ν-vector stay put in a processor ($\nu_{k,j}$ resides in processor j for all k) while the components of the μ-vector are shifted one processor to the right in each step.

Suppose a third array for backsubstitution is available whose processors are connected to the corresponding processors of the factorisation array. Since $\nu_{k,j-k-1}$ is computed in processor $j-k-1$ at time $\tau = 4n+3-k-j$ it can be used at time $4n+5$ in processor $j-k$ of the backsubstitution array. With processor and time functions

$$(\pi_1 \quad \pi_2 \quad \pi_3) = (-1 \quad 1 \quad 0), \quad (\tau_1 \quad \tau_2 \quad \tau_3) = (-1 \quad -1 \quad 4n+5),$$

$x_{j,k}$ can be computed in processor $j-k$ at time $4n+5-k$. Since processor 0 has retained d_k from the previous re-generation phase and g_k is computed early enough in processor 0 of the forward elimination array (at time $\tau = 2k+1$), element $x_k = x_{k,k}$ of the solution vector can be computed in processor 0 at time $4n+5-2k$. The whole computation is completed in $4n+6$ time steps. Note that during step k of the factorisation and forward elimination phase there are $2k$ idle processors.

Therefore, $6n$ processors can compute the solution to a $n \times n$ Toeplitz system in time $O(4n)$ only relying on nearest neighbour computation, however the storage in at least one processor must be proportional to the problem size $O(n)$.

Backsubstitution

The last step is normally solved by backsubstitution $x = L_n^T D_n^{-1} g$ without making any use of the Toeplitz structure of the original system. A new approach that uses the Schur recursions also for the last step was derived in [7] and can be related to the Levinson recursions as follows.

Remember that $((\psi_{n-k}^T \quad 1) \quad 0_k^T)$, is the kth column of L_n^T, that $g_k = \alpha_{k,k}$ is the kth element of the vector g, and that d_k is the kth diagonal element of D_n, $0 \le k \le n$. With the abbreviation $\gamma_k = g_k/d_k$, $0 \le k \le n$, one can express the solution vector as a linear combination of the columns

of L_n^T:

$$x = \gamma_0 \begin{pmatrix} 1 \\ 0_n \end{pmatrix} + \gamma_1 \begin{pmatrix} \psi_1 \\ 1 \\ 0_{n-1} \end{pmatrix} + \gamma_2 \begin{pmatrix} \psi_2 \\ 1 \\ 0_{n-2} \end{pmatrix} + \ \ldots \ + \gamma_{n-1} \begin{pmatrix} \psi_{n-1} \\ 1 \\ 0 \end{pmatrix} + \gamma_n \begin{pmatrix} \psi_n \\ 1 \end{pmatrix}.$$

Define the partial sums

$$x^{(n)} = \gamma_n \begin{pmatrix} \psi_n \\ 1 \end{pmatrix}, \qquad x^{(n-k-1)} = x^{(n-k)} + \gamma_{n-k-1} \begin{pmatrix} \psi_{n-k-1} \\ 1 \\ 0_{k+1} \end{pmatrix}, \quad 0 \le k \le n-1,$$

so that $x^{(0)} = x$. It will now be shown by induction that for $1 \le k \le n$

$$x^{(n-k)} = \begin{pmatrix} 0_{n-k} \\ \epsilon_{n-k,n-k} \\ \vdots \\ \epsilon_{n-k,n} \end{pmatrix} + \begin{pmatrix} \eta_{n-k,n-k} \\ \vdots \\ \eta_{n-k,n} \\ 0_{n-k} \end{pmatrix} + \sum_{j=0}^{k} \epsilon_{n-k,n-j} \begin{pmatrix} 0_{k-j+1} \\ \psi_{n-k-1} \\ 0_{j+1} \end{pmatrix} + \sum_{j=0}^{k} \eta_{n-k,n-j} \begin{pmatrix} 0_{k-j+1} \\ J\psi_{n-k-1} \\ 0_{j+1} \end{pmatrix}. \quad (9)$$

(i) For $k = 0$ it follows from the Levinson recursion (4) that

$$x^{(n)} = \gamma_n \begin{pmatrix} \psi_n \\ 1 \end{pmatrix} = \gamma_n \left\{ \begin{pmatrix} 0 \\ \psi_{n-1} \\ 1 \end{pmatrix} + \rho_n \begin{pmatrix} 1 \\ J\psi_{n-1} \\ 0 \end{pmatrix} \right\}$$

$$= \begin{pmatrix} 0_n \\ \epsilon_{n,n} \end{pmatrix} + \begin{pmatrix} \eta_{n,n} \\ 0_n \end{pmatrix} + \epsilon_{n,n} \begin{pmatrix} 0 \\ \psi_{n-1} \\ 0 \end{pmatrix} + \eta_{n,n} \begin{pmatrix} 0 \\ J\psi_{n-1} \\ 0 \end{pmatrix},$$

where $\epsilon_{n,n} = \gamma_n = g_n/d_n$ and $\eta_{n,n} = \rho_n \gamma_n$.

(ii) Assume the statement is true for $k \ge 0$.

(iii) Using the induction hypothesis (ii) for $x^{(n-k)}$ in

$$x^{(n-k-1)} = x^{(n-k)} + \gamma_{n-k-1} \begin{pmatrix} \psi_{n-k-1} \\ 1 \\ 0_{k+1} \end{pmatrix}$$

and making use of the Levinson recursion in each of the two sums of (9) results in

$$\sum_{j=0}^{k} \epsilon_{n-k,n-j} \begin{pmatrix} 0_{k-j+1} \\ \psi_{n-k-1} \\ 0_{j+1} \end{pmatrix} = \sum_{j=0}^{k} \epsilon_{n-k,n-j} \left\{ \begin{pmatrix} 0_{k-j+1} \\ 0 \\ \psi_{n-k-2} \\ 0_{j+1} \end{pmatrix} + \rho_{n-k-1} \begin{pmatrix} 0_{k-j+1} \\ 1 \\ J\psi_{n-k-2} \\ 0_{j+1} \end{pmatrix} \right\}$$

$$= \epsilon_{n-k,n} \left\{ \begin{pmatrix} 0_{k+1} \\ 0 \\ \psi_{n-k-2} \\ 0 \end{pmatrix} + \rho_{n-k-1} \begin{pmatrix} 0_{k+1} \\ 1 \\ J\psi_{n-k-2} \\ 0 \end{pmatrix} \right\} + \sum_{j=0}^{k-1} \epsilon_{n-k,n-j-1} \left\{ \begin{pmatrix} 0_{k-j} \\ 0 \\ \psi_{n-k-2} \\ 0_{j+2} \end{pmatrix} + \rho_{n-k-1} \begin{pmatrix} 0_{k-j} \\ 1 \\ J\psi_{n-k-2} \\ 0_{j+2} \end{pmatrix} \right\}$$

for the first sum and

$$\sum_{j=0}^{k} \eta_{n-k,n-j} \begin{pmatrix} 0_{k-j+1} \\ J\psi_{n-k-1} \\ 0_{j+1} \end{pmatrix} = \sum_{j=0}^{k} \eta_{n-k,n-j} \left\{ \begin{pmatrix} 0_{k-j+1} \\ J\psi_{n-k-2} \\ 0 \\ 0_{j+1} \end{pmatrix} + \rho_{n-k-1} \begin{pmatrix} 0_{k-j+1} \\ \psi_{n-k-2} \\ 1 \\ 0_{j+1} \end{pmatrix} \right\}$$

$$= \sum_{j=0}^{k-1} \eta_{n-k,n-j} \left\{ \begin{pmatrix} 0_{k-j+1} \\ J\psi_{n-k-2} \\ 0 \\ 0_{j+1} \end{pmatrix} + \rho_{n-k-1} \begin{pmatrix} 0_{k-j+1} \\ \psi_{n-k-2} \\ 0 \\ 0_{j+1} \end{pmatrix} \right\} + \eta_{n-k,n-k} \left\{ \begin{pmatrix} 0 \\ J\psi_{n-k-2} \\ 0 \\ 0_{k+1} \end{pmatrix} + \rho_{n-k-1} \begin{pmatrix} 0 \\ \psi_{n-k-2} \\ 1 \\ 0_{k+1} \end{pmatrix} \right\}$$

for the second sum. The last term in $x^{(n-k-1)}$ expands to

$$\gamma_{n-k-1} \begin{pmatrix} \psi_{n-k-1} \\ 1 \\ 0_{k+1} \end{pmatrix} = \gamma_{n-k-1} \left\{ \begin{pmatrix} 0 \\ \psi_{n-k-2} \\ 1 \\ 0_{k+1} \end{pmatrix} + \rho_{n-k-1} \begin{pmatrix} 1 \\ J\psi_{n-k-2} \\ 0 \\ 0_{k+1} \end{pmatrix} \right\}.$$

Collecting corresponding terms gives the following expression for $x^{(n-k-1)}$

$$\begin{pmatrix} 0_{n-k} \\ \epsilon_{n-k,n-k} \\ \vdots \\ \epsilon_{n-k,n} \end{pmatrix} + \begin{pmatrix} \eta_{n-k,n-k} \\ \vdots \\ \eta_{n-k,n} \\ 0_{n-k} \end{pmatrix} + \begin{pmatrix} 0_{k+2} \\ \epsilon_{n-k,n}\psi_{n-k-2} \\ 0 \end{pmatrix} + \begin{pmatrix} 0_{k+1} \\ \rho_{n-k-1}\epsilon_{n-k,n} \\ \rho_{n-k-1}\epsilon_{n-k,n}J\psi_{n-k-2} \\ 0 \end{pmatrix}$$

$$+ \sum_{j=0}^{k-1} \begin{pmatrix} 0_{k-j} \\ 0 \\ (\epsilon_{n-k,n-j-1} + \rho_{n-k-1}\eta_{n-k,n-j})\psi_{n-k-2} \\ \rho_{n-k-1}\eta_{n-k,n-j} \\ 0_{j+1} \end{pmatrix} + \sum_{j=0}^{k-1} \begin{pmatrix} 0_{k-j} \\ \rho_{n-k-1}\epsilon_{n-k,n-j-1} \\ (\eta_{n-k,n-j-1} + \rho_{n-k-1}\epsilon_{n-k,n-j-1})J\psi_{n-k-2} \\ 0 \\ 0_{j+1} \end{pmatrix}$$

$$+ \begin{pmatrix} 0 \\ (\gamma_{n-k-1} + \rho_{n-k-1}\eta_{n-k,n-k})\psi_{n-k-2} \\ \gamma_{n-k-1} + \rho_{n-k-1}\eta_{n-k,n-k} \\ 0_{k+1} \end{pmatrix} + \begin{pmatrix} \rho_{n-k-1}\gamma_{n-k-1} \\ (\eta_{n-k,n-k} + \rho_{n-k-1}\gamma_{n-k-1})J\psi_{n-k-2} \\ 0_{k+2} \end{pmatrix}$$

which can be written as

$$\begin{pmatrix} 0_{n-k-1} \\ \epsilon_{n-k-1,n-k-1} \\ \vdots \\ \epsilon_{n-k-1,n} \end{pmatrix} + \begin{pmatrix} \eta_{n-k-1,n-k-1} \\ \vdots \\ \eta_{n-k-1,n} \\ 0_{n-k-1} \end{pmatrix} + \sum_{j=0}^{k+1} \begin{pmatrix} 0_{j-k+1} \\ \epsilon_{n-k-1,n-j}\psi_{n-k-2} \\ 0_{j+1} \end{pmatrix} + \sum_{j=0}^{k+1} \begin{pmatrix} 0_{j-k+1} \\ \eta_{n-k-1,n-j}J\psi_{n-k-2} \\ 0_{j+1} \end{pmatrix},$$

where

$$\begin{pmatrix} \epsilon_{n-k-1,n-k-1} & \epsilon_{n-k-1,n-k} & \cdots & \epsilon_{n-k-1,n-1} & \epsilon_{n-k-1,n} \\ \eta_{n-k-1,n-k-1} & \eta_{n-k-1,n-k} & \cdots & \eta_{n-k-1,n-1} & \eta_{n-k-1,n} \end{pmatrix}$$
$$= \begin{pmatrix} 1 & \rho_{n-k-1} \\ \rho_{n-k-1} & 1 \end{pmatrix} \begin{pmatrix} \gamma_{n-k-1} & \epsilon_{n-k,n-k} & \cdots & \epsilon_{n-k,n-1} & \epsilon_{n-k,n} \\ \eta_{n-k,n-k} & \eta_{n-k,n-k+1} & \cdots & \eta_{n-k,n} & 0 \end{pmatrix}.$$

This completes the induction.

The backsubstitution part using the Schur recursions computes the ϵ- and η-vectors and can be formulated as follows.

Backsubstitution by Schur Recursions

The Schur recursions determine the vector $x = L_n^T D_n^{-1} g$ with its kth element given by x_k.

$$\begin{pmatrix} \epsilon_{n,n} \\ \eta_{n,n} \end{pmatrix} = \begin{pmatrix} g_n/d_n \\ 0 \end{pmatrix}$$

$$1 \le k \le n, \quad \begin{pmatrix} \epsilon_{n-k,n-k} & \epsilon_{n-k,n-k+1} & \cdots & \epsilon_{n-k,n-1} & \epsilon_{n-k,n} \\ \eta_{n-k,n-k} & \eta_{n-k,n-k+1} & \cdots & \eta_{n-k,n-1} & \eta_{n-k,n} \end{pmatrix}$$
$$= \begin{pmatrix} 1 & \rho_{n-k} \\ \rho_{n-k} & 1 \end{pmatrix} \begin{pmatrix} g_{n-k}/d_{n-k} & \epsilon_{n-(k-1),n-(k-1)} & \cdots & \epsilon_{n-(k-1),n-1} & \epsilon_{n-(k-1),n} \\ \eta_{n-(k-1),n-(k-1)} & \eta_{n-(k-1),n-(k-1)+1} & \cdots & \eta_{n-(k-1),n} & 0 \end{pmatrix}$$

$$0 \le j \le n, \quad x_j = \epsilon_{0,j} + \eta_{0,j}.$$

Third Class of Systolic Implementations

The computation of all phases, factorisation, forward elimination and backsubstitution, by Schur recursions makes it possible to employ only one array for all three phases. The corresponding array in [7, 8] is the most efficient of the three types of designs presented, and can be derived as follows (the processor and time functions here differ from the ones in [7, 8] in a few small details that do not affect the asymptotic computation time).

To fit all three phases on one array it is convenient to adapt the index structure of the factorisation phase to that of forward elimination by transforming each index $(k \quad j)$ in the factorisation to

$$(k \quad j)\begin{pmatrix}1 & 1\\ 0 & 1\end{pmatrix} = (k \quad k+j).$$

The transformed factorisation phase is thus expressed as:

$$d_0 = t_0$$

$$\begin{pmatrix}\nu_{0,0} & \cdots & \nu_{0,n-1}\\ \mu_{0,0} & \cdots & \mu_{0,n-1}\end{pmatrix} = \begin{pmatrix}t_1 & \cdots & t_n\\ t_1 & \cdots & t_n\end{pmatrix}$$

$$1 \le k \le n, \quad \rho_k = -\mu_{k-1,k-1}/d_{k-1}, \quad d_k = d_{k-1}(1-\rho_k^2)$$

$$\begin{pmatrix}\nu_{k,k} & \cdots & \nu_{k,n-1}\\ \mu_{k,k} & \cdots & \mu_{k,n-1}\end{pmatrix} = \begin{pmatrix}1 & \rho_k\\ \rho_k & 1\end{pmatrix}\begin{pmatrix}\nu_{k-1,k-1} & \cdots & \nu_{k-1,n-2}\\ \mu_{k-1,k} & \cdots & \mu_{k-1,n-1}\end{pmatrix}.$$

The time and processor functions are chosen to be

$$(\pi_1 \quad \pi_2 \quad \pi_3) = (1 \quad 0 \quad 0), \quad (\tau_1 \quad \tau_2 \quad \tau_3) = (1 \quad 1 \quad 2).$$

Thus, all matrix elements are input to the same processor: t_k is input to processor 0 at time $\tau = k+1$; and $(\nu_{k,j} \quad \mu_{k,j})$ are determined in processor k at time $\tau = k+j+2$. The values of ρ_k and d_k are computed along with $\nu_{k,k}$, in processor k at time $2k+2$, and remain in that processor throughout factorisation and forward elimination. Notice that the components of the ν-vector stay put in the processor while the components of the μ-vector are shifted one processor to the left. The last quantities ρ_n and d_n are computed in processor n at time $2(n+1)$, so the computation of the factorisation requires $2n+3$ steps.

Since the factorisation has the same structure as the forward elimination phase, and the forward elimination phase involves the ρ_k which are now computed in different processors the two phases may be overlapped, thereby eliminating the processor idle time of the previous designs. Observe that the last matrix element t_n is input to processor 0 at $\tau = n+2$ so the first element of the right-hand side vector f_0 can be input to processor 0 at time $n+3$. In general, all right-hand side elements are input to the same processor as the matrix elements: f_j is input to processor 0 at time $n+j+3$, and time and processor functions (except for the time displacement τ_3) are the same as before:

$$(\pi_1 \quad \pi_2 \quad \pi_3) = (1 \quad 0 \quad 0), \quad (\tau_1 \quad \tau_2 \quad \tau_3) = (1 \quad 1 \quad n+3).$$

The pair $(\alpha_{k,j} \quad \beta_{k,j})$ is determined in processor k at time $\tau = k+j+n+3$, and the components of both α- and β-vectors experience a shift to the left neighbouring processor after their computation. Element $g_k = \alpha_{k,k}$ is computed in processor k at time $2k + n + 3$, and forward elimination is completed at time $3n + 4$.

To keep communication on a nearest neighbour basis, the linear array is folded together so that processors k and $n - k$ are situated across from each other. After completion of the forward elimination phase processors k and $n - k$ can then exchange their values of ρ, d and g so that processor k ends up with ρ_{n-k}, d_{n-k} and g_{n-k}. For simplicity each index $(k \quad j)$ of backsubstitution is transformed to

$$(k \quad j)\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} + (n \quad 0) = (n-k \quad j),$$

resulting in

$$\begin{pmatrix} \epsilon_{0,n} \\ \eta_{0,n} \end{pmatrix} = \begin{pmatrix} g_n/d_n \\ 0 \end{pmatrix}$$

$$1 \le k \le n, \quad \begin{pmatrix} \epsilon_{k,n-k} & \epsilon_{k,n-k+1} & \cdots & \epsilon_{k,n-1} & \epsilon_{k,n} \\ \eta_{k,n-k} & \eta_{k,n-k+1} & \cdots & \eta_{k,n-1} & \eta_{k,n} \end{pmatrix}$$

$$= \begin{pmatrix} 1 & \rho_{n-k} \\ \rho_{n-k} & 1 \end{pmatrix} \begin{pmatrix} g_{n-k}/d_{n-k} & \epsilon_{k-1,n-(k-1)} & \cdots & \epsilon_{k-1,n-1} & \epsilon_{k-1,n} \\ \eta_{k-1,n-(k-1)} & \eta_{k-1,n-(k-1)+1} & \cdots & \eta_{k-1,n} & 0 \end{pmatrix}$$

$$0 \le j \le n, \quad x_j = \epsilon_{n,j} + \eta_{n,j}.$$

With processor and time functions

$$(\pi_1 \quad \pi_2 \quad \pi_3) = (1 \quad 0 \quad 0), \quad (\tau_1 \quad \tau_2 \quad \tau_3) = (2 \quad 1 \quad 2n+4)$$

the pair $(\epsilon_{k,j} \quad \eta_{k,j})$ is computed on processor k at time $\tau = 2k+j+2n+4$. In particular, component $x_k = \epsilon_{n,k} + \eta_{n,k}$ of the solution vector is computed in processor n at time $\tau = 4(n+1) + k$. Hence backsubstitution is completed at time $5n + 6$.

With the above scheme, a Toeplitz system of order n can be solved in time $O(5n)$ on n processors with nearest neighbour commuincation. Each processor requires only a constant amount of storage. External input takes place on the first processor and external output on the last. As shown in [7, 8] the solution processes for several different problems with different right-hand sides can be overlapped and the solution to a new problem can be obtained every n steps. Furthermore, as shown in [8], the above array belongs to the class of n-processor arrays that solve Toeplitz systems faster than any other array with linear processor and time function, and I/O restricted to the boundary processors.

References

[1] Bareiss, E.H., *Numerical Solution of Linear Equations with Toeplitz and Vector Toeplitz Matrices,* Numer. Math., 13 (1969), pp. 404–24.

[2] Brent, R.P., Gustavson, F.G. and Yun, D.Y.Y., *Fast Solution of Toeplitz Systems of Equations and Computation of Pade Approximants,* J. Algorithms, 1 (1980), pp. 159–95.

[3] Brent, R.P., Kung, H.T. and Luk, F.T., Some Linear-Time Algorithms for Systolic Arrays, *Proc. IFIP 9th World Computer Congress,* North Holland, Amsterdam, 1983, pp. 865–76.

[4] Brent, R.P. and Luk, F.T., *A Systolic Array for the Linear-Time Solution of Toeplitz Systems of Equations,* J. VLSI and Computer Systems, 1 (1983), pp. 1–22.

[5] Bunch, J.R., *Stability of Methods for Solving Toeplitz Systems of Equations,* SIAM J. Sci. Stat. Comput., 6 (1985), pp. 349–64.

[6] Delosme, J.-M., *Algorithms for Finite Shift-Rank Processes,* Ph.D. Thesis, Dept of Electrical Engineering, Stanford University, 1982.

[7] Delosme, J.-M. and Ipsen, I.C.F., *Parallel Solution of Symmetric Positive Definite Systems with Hyperbolic Rotations,* Linear Algebra and its Applications, 77 (1986), pp. 75–111.

[8] ————, Efficient Systolic Arrays for the Solution of Toeplitz Systems : An Illustration of a Methodology for the Construction of Systolic Architectures in VLSI, *Systolic Arrays,* Adam Hilger, 1987, pp. 37–46.

[9] Delosme, J.-M. and Morf, M., Normalized Doubling Algorithms for Finite Shift-Rank Processes, *Proc. 20th IEEE Conference on Decision and Control,* 1981, pp. 246–8.

[10] Grenander, U. and Szego, G., *Toeplitz Forms and their Applications,* University of California Press, 1958.

[11] Iohvidov, I.S., *Hankel and Toeplitz Matrices and Forms,* Birkhauser, 1982.

[12] Kung, S.-Y. and Hu, Y.H., *A Highly Concurrent Algorithm and Pipelined Architecture for Solving Toeplitz Systems,* IEEE Trans. Acoustics, Speech, and Signal Processing, ASSP-31 (1983), pp. 66–76.

[13] Kung, H.T. and Leiserson, C.E., Systolic Arrays (for VLSI), *Sparse Matrix Proceedings,* SIAM, Philadelphia, PA, 1978, pp. 256–82.

[14] Levinson, N., *The Wiener RMS (Root-Mean-Square) Error Criterion in Filter Design and Prediction,* J. Math. Phys., 25 (1947), pp. 261–78.

[15] Morf, M., *Fast Algorithms for Multivariable Systems,* Ph.D. Thesis, Dept of Electrical Engineering, Stanford University, 1974.

[16] Nash, J.G., Hansen, S. and Nudd, G.R., VLSI Processor Array for Matrix Manipulation, *CMU Conference on VLSI Systems and Computations,* Computer Science Press, 1981, pp. 367–73.

[17] Roebuck, P.A. and Barnett, S., *A Survey of Toeplitz and Related Matrices,* Int. J. Systems Sci., 9 (1978), pp. 921–34.

[18] Schur, I., *Ueber Potenzreihen die im Innern des Einheitskreises Beschraenkt Sind,* J. Reine Angewandte Mathematik, 147 (1917), pp. 205–32.

[19] Trench, W.F., *An Algorithm for The Inversion of Finite Toeplitz Matrices,* J. Soc. Indust. Appl. Math., 12 (1964), pp. 515–22.

END

8-87

DTIC